THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Low-Level Attacks in Bitcoin Wallets

Citation for published version:
Gkaniatsou, A, Arapinis, M & Kiayias, A 2017, Low-Level Attacks in Bitcoin Wallets. in *Information Security : 20th International Conference, ISC 2017, Ho Chi Minh City, Vietnam, November 22-24, 2017, Proceedings.* Lecture Notes in Computer Science (LNCS), vol. 10599, Springer, Cham, pp. 233-253, 20th International Information Security Conference, Ho Chi Minh City, Viet Nam, 22/11/17. https://doi.org/10.1007/978-3-319-69659-1_13

Digital Object Identifier (DOI):
10.1007/978-3-319-69659-1_13

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Information Security

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 07. May. 2023

Low-Level Attacks in Bitcoin Wallets

Andriana Gkaniatsou[1], Myrto Arapinis[2], and Aggelos Kiayias[3]

School of Informatics, University of Edinburgh, UK
[1]a.e.gkaniatsou@sms.ed.ac.uk, [2]marapini@inf.ed.ac.uk, [3]aggelos.kiayias@ed.ac.uk

Abstract. As with every financially oriented protocol, there has been a great interest in studying, verifying, attacking, identifying problems, and proposing solutions for Bitcoin. Within that scope, it is highly recommended that the keys of user accounts are stored offline. To that end, companies provide solutions that range from paper wallets to tamper-resistant smart-cards, offering different level of security. While incorporating expensive hardware for the wallet purposes is though to bring guarantees, it is often that the low-level implementations introduce exploitable back-doors. This paper aims to bring to attention how the overlooked low-level protocols that implement the hardware wallets can be exploited to mount Bitcoin attacks. To demonstrate that, we analyse the general protocol behind LEDGER Wallets, the only EAL5+ certified against side channel analysis attacks hardware. In this work we conduct a throughout analysis on the Ledger Wallet communication protocol and show how to successfully attack it in practice. We address the lack of well-defined security properties that Bitcoin wallets should conform by articulating a minimal threat model against which any hardware wallet should defend. We further use that threat model to propose a lightweight fix that can be adopted by different technologies.

1 Introduction

Bitcoin is currently considered to be the most successful cryptocurrency, with an estimated average daily transaction value of US$200K. As it is becoming the most widely adopted digital currency, there is substantial resource and research investment into the security of the Bitcoin protocol and its transactions. Bitcoin is based on public key cryptography, which requires users to digitally sign their payments to prove ownership. Therefore, a salient aspect of Bitcoin is the wallet key management: loss of the private keys effectively means loss of funds; exposure of the public keys conveys privacy loss.

Online wallets are popular with Bitcoin users, as they are offered as a service that is faster and safer than running the Bitcoin client locally. User accounts are hosted on remote servers and accessed through third-party Web services; wallets either store the keys also in remote servers, or locally in the user's web client (typically a web browser). The user acceses his wallet through web-based authentication mechanisms and all cryptographic operations take place server-side, typically in the Cloud. Although this approach is popular among Bitcoin users, certain security issues arise as the user's private keys can be exploited by the host. For instance, in 2013 the StrongCoin web-hosted wallet transferred without user consent bitcoins from their servers to a different service, OzCoin, as it was claimed to be stolen [7]. Online wallets are also common targets for

Distributed Denial of Service (DDoS) attacks, *e.g.*, BitGo and `blockchain.info` in June 2016. Such examples raised concerns about the reliability of such wallets and created the trend for *cold storage* and *cryptographic tokens*, with most major companies having integrated their software wallets with hardware devices.

wallet	secure element	HID	encrypted channel
LEDGER *HW.1*	smart card	×	×
LEDGER *Nano*	smart card	×	×
LEDGER *Nano S*	smart card	✓	×
Trezor	microcontroller	✓	×
KeepKey	microcontroller	✓	×
Digital BitBox	microcontroller	✓	✓

Table 1: Bitcoin Hardware Wallets Characteristics

Hardware wallets aim to offer a secure environment for key management and transaction signing. When a user requests a payment, the wallet's API creates the corresponding Bitcoin transaction and sends it to the hardware to be signed. The hardware signs the transaction and returns the signature together with the corresponding public key to the API, which is then pushed it to the network. In that way the senstive signing keys do not ever leave the secure environment of the hardware wallet. The Bitcoin wallets currently in the market incorporate either microcontrollers or smart-cards. As of April 2017, the hardware wallet options suggested by `bitcoin.org` are the three LEDGER wallets, which are based on smart-cards; or *Trezor*, *Digital Bitbox* and *Keepkey*, which are based on microcontrollers. All wallets offer two versions: (*a*) a plain USB dongle, or (*b*) a USB Human Interface Device (HID) with an embedded screen for the user to verify and confirm the transaction. The main differences between current hardware wallets are shown in Table 1. Currently, apart from Digital BitBox, none of the wallets uses a secure communication channel. Offering a tamper resilient cryptographic memory is not enough on its own to guarantee against transaction attacks. Unauthorised access to the signing oracle of the wallet is not much different from plain access to the keys themselves, as both allow the funds to be stolen. Processing a Bitcoin request involves the communication between the hardware wallet and third-party systems. The lack of a general threat model for the Bitcoin wallets and well-defined specifications of that communication leads to proprietary implementations. As previous studies on different protocols have shown (*e.g.*, [8, 10, 13]), such practice often results in insecure low-level implementations that are prone to *Man-in-the-Middle* (MitM) attacks.

All hardware wallets implement a payment protocol similar to the following. The API broadcasts to the device the input funds and the payment details and requests the transaction signature If the device supports a second factor verification mechanism for the payment, it will sign the transaction only after the user's approval. If the device does not support such mechanism, it will sign it immediately. Although most Bitcoin wallets claim to secure the transactions by enforcing the user's validation of the payment data, the success rate of transaction attacks is analogous to the user error rate. The validation/comparison of hashes by the user, is a common technique *e.g.*, device pairing, self-signed certificates with HTTPS *etc.*. The usability aspects of hash comparison in security protocols and the effects of human errors have been studied before. For example, in [30] the authors conclude that the compare-and-confirm method (the user has to confirm a checksum presented on the device's screen) for a 4-digit string has 20% fail-

ure rate, whereas the work in [17] concludes that comparison of the Base32[1] hashes has an average 14% failure rate. Such studies focus on low entropy hashes and suggest that raising the entropy would result in bigger error rates. They conclude such techniques cannot provide strong security guarantees. Thus, a transaction attack on an HID wallet depends on the user's ability to identify the tampered data.

In this paper we stress the importance of securing the low-level communication of Bitcoin hardware wallets. We show that by taking advantage of that communication layer it is possible to propagate the attacks directly to the underlying Bitcoin transactions. The attacks we address are general and target any low-level communication with hardware wallets. Applying them in practice is a matter of adapting them to the corresponding hardware implementation. The security of microcontrollers has been extensively examined, and a number of fault and side-channel attacks have been found, *e.g.*, [4, 12, 20, 21]. Therefore, we focus on smart-card based wallets, which provide guarantees against physical and interdiction attacks and have traditionally been used for key management and cryptographic operations. As of April 2017, LEDGER is the only company offering smart-card solutions. The LEDGER wallets are EAL5+ certified and are advertised as the most secure, tamper-proof and trustworthy devices for managing Bitcoin transactions.[2]

We consider client-side security and not security in the Bitcoin network, although a single wallet attack may escalate. Attacking Bitcoin at the network level is immensively expensive as it requires great computational resources. General attacks on Bitcoin wallets that could be applied to several users simultaneously are a much cheaper, easier and efficient way to gain access to multiple accounts. The LEDGER API is available on the *Chrome Web Store*, making it the ideal target for massively attacking users.

Our Contributions and Roadmap. To the best of our knowledge our work is the first to: (i) stress the importance of securing Bitcoin transactions and preserving the account's privacy at the wallet level, (ii) consider a minimal threat model for hardware Bitcoin wallets, and (iii) address the security issues originating in low-level communication of Bitcoin devices, by showcasing practical attacks. We provide a thorough analysis of the LEDGER wallets by extracting their protocols, analysing them and showing practical attacks. We propose a lightweight and user-friendly fix which is general enough to be adapted to all wallets regardless the hardware technology. As, the LEDGER protocols are not publicly available, we reverse-engineered the communication protocol and abstracted its implementations. In Section 3 we present and analyse the protocols that we extracted. In Section 4 we articulate a general purpose threat model for Bitcoin wallets and show how we have successfully mounted the identified attacks on LEDGER wallets. To that end, in Section 4 we propose a lightweight and easily adaptable fix that requires minimal changes.

[1] Base32 hashes are a total of 25 bit entropy and consist of five characters with 32 possible character mappings. A Bitcoin address has 160 bit entropy.

[2] See `http://goo.gl/KhtWXc`, `http://goo.gl/sbYXzh`, `http://goo.gl/hOU5jB`.

2 Background

Bitcoin is a *Peer-to-Peer* (P2P) payment system that utilises public-key cryptography and consists of addresses and transactions. A transaction may have multiple inputs and outputs and is formed by digitally signing the hash of the transaction from which specific funds are transfered. The signature and the corresponding public key are sent to the network for verification. Upon successful validation, the funds are transferred to the stated addresses. Assuming a user u with a private/public key pair (sk_u, pk_u), let x_u be the recipient address, generated by hashing pk_u; let y_u be the hash of transaction t_u that transferred the funds to x_u. The transaction that further transfers b funds to some address z_p is the signature Sig_u of y_u, b and z_p using private key sk_u: $Sig_{sk_u}(y_u, b, z_p)$. Once a transaction is formed, it is broadcast to the network to be validated for: (*a*) outputs not exceeding inputs, (*b*) the user's ownership of the funds by verification of the signature with the corresponding pk_u.

Transactions in Bitcoin are expressed in a scripting language known as the *Bitcoin raw* protocol, which defines the conditions on the inputs and the outputs. According to [6], a transaction is defined in blocks of bytes. Table 2 presents the specific structure of a transaction block and the abbreviations that we will use in the next Sections: v is a fixed constant that defines the block format version; ic is a counter for the inputs; $txid_i$ is the reference to the previous transaction whose outputs will fund the current transaction; pc is a reference to the outputs of $txid_i$ that will be used; $sigL$ is the length of the signature; $scriptSig$ is the signature of the current transaction with the private key that correspond to the previous transaction outputs; s is a fixed constant that defines the end of the inputs declaration; oc is a counter for the outputs of the current transaction; $amount_t$ corresponds to the amount to be spent and l to the length of the destination public key; $addr_p$ is the recipient public key for $amount_t$.

	v: version	4 bytes
inputs	ic: input count	1 byte
	$txid_i$: previous transaction id (hash)	variable length
	pc: previous output index	4 bytes
	$sigL$: script signature length	1 byte
	$scriptSig$: script signature	variable length
	s: sequence	4 bytes
outputs	oc: output count	1 byte
	$amount_t$: value	8 bytes
	l: script length	1 byte
	$addr_p$: scriptPubKey	variable length
	bt: block lock time	4 bytes

Table 2: The Transaction Block.

Upon payment, the wallet must access the previous transactions and the available funds. Memory limitations and absence of access to the network, make it difficult for hardware wallets to track previous transactions. Segregated Witnesses (SegWit) solve that problem by including the value of the inputs in the signature of the transaction: hardware wallets then hash the inputs and sign that hash.

Key Management of Hardware Wallets. Currently all hardware wallets implement a *Hierarchical Deterministic* (HD) wallet of BIP32, which generates a new key-pair for each address request [32]. HD wallets derive fresh private keys from a common master key pair $\{sk_m, pk_m\}$. For the creation of a new wallet a 128- to 512-bit seed s, a sequence of random numbers, is generated. The master private key sk_m is gen-

erated by a function $sk_m = hash(s)$ where $hash(s)$ is the SHA256 hash of s. Then, given the master key pair (sk_m, pk_m), the wallet generates and maintains a sequence of children private $sk_1, sk_2, ...$ and public $pk_1, pk_2, ..$ keys from the master private key sk_m. A key sk_i is derived by the function $sk_i = sk_m + hash(i, pk_m)\ (mod\ n)$, $pk_i = pk_m + hash(i, pk_m)N$ or equally $sk_i N$ with i denoting the index of the key, and $hash$ being the HMAC-SHA512 function. Children public keys pk_i can be derived only by knowing the master public key pk_m and the index i.

Related Work. Previous work on attacking Bitcoin has exposed malleability attacks, where the adversary forces the victims to generate a transaction to an address controlled by her. When a victim broadcasts the transaction to the network, the adversary obtains a copy of that transaction that she modifies by tampering the signature without invalidating it. That modification results in a different transaction identifier (hash). The adversary then broadcasts the tampered transaction to the network, resulting in the same transaction being in the network under two different hashes. As a single transaction can only be confirmed once, only one of these two transactions will be included in a block and the other will be ignored. The attack is successful if the attacker's modified version is accepted. Although this attack is not new, it was given great attention after the malleability attack on MtGox [11], the first and one of the largest Bitcoin exchanges, in 2014. Since then different malleability attacks and solutions have been proposed, *e.g.*, [11,31]. Double spending is another class of attacks on Bitcoin transactions, where the user spends the same coin twice. The feasibility of double spending attacks by using hashrate-based attack models was studied in [24, 26]. It was shown that the attack is successful whenever the number of confirmations of a dishonest transaction is greater than the number of confirmations of the honest one. In [19] the authors exploit non-confirmed transactions to implement double spending attacks on fast payments, and [26] shows how such attacks coupled with high computational resources can have a higher success rate. Apart from the attacks that target transactions, privacy has also been targeted. Though privacy is a concern of the original specification [24] the public nature of Bitcoin renders strong privacy difficult to achieve. For instance, by tracing the flow of coins it is possible to identify their owner [15]. Likewise, [1] studied how transaction behaviour can be linked with a single account.

All the aforementioned attacks do not tackle the wallet layer. They all assume the wallet implementation to be secure. As many malware attacks have gained publicity *e.g.*, [5, 18, 25] or the malware attack on the Bitstamp wallet that costed US$5M [16], the importance of protecting Bitcoin wallets has been repeatedly stressed out [28]. [3] proposes a *super-wallet* as a solution to malware, in which the funds are split across multiple devices using cryptographic threshold techniques. The importance of ensuring wallet security is also presented in [29] where the authors formally analyse the authentication properties of the *Electrum* wallet. The authors of [22] and [2] argue that Bitcoin wallets be tamper-resistant and propose cryptographic tokens as a countermeasure to malware attacks. Our work exploits Bitcoin transactions at the wallet level. Instead of attacking the Bitcoin raw protocol directly, we show the importance of the protocols connected to the Bitcoin implementations. Attacking such protocols overrides any security restrictions that expensive hardware additions may add, and can be equally harmful to attacking the Bitcoin raw protocol itself.

3 Ledger Wallet Implementation

The low-level communication layer of LEDGER wallets, defined by the APDU layer, is crafted to implement the Bitcoin raw protocol. The communication consists of a series of raw hexadecimal command-response pairs between the API and the hardware: the API retrieves data or requests the hardware to execute a specific operation via APDU commands; whereas the hardware responds to that request via APDU responses. For example, in the following sequence:

command	`e04800001f058000002c8000000080000000000000000000000c040406060200 00000001`
response	`3044022033128d0d576487e2e0c5892c0915564a6a5f119e698c033262d66052 7943a16d022009caa037703d9a3dbf7eec4cecca08bf33b3b9a18ef929a810f8 faf6ab0f1c7a01`

command retrieves the signature (*response*) over some transaction data. The LEDGER protocols are closed-source and there does not exist any information on how the Bitcoin specifications are translated into the APDU layer. A large part of our work has been to reverse-engineer the APDU layer and extract the implemented protocol. This was achieved by creating a man-in-the-middle sniffer[3] sitting on top of the Ledger API, capable of recording and interfering with the communication during any active sessions with the dongle. To abstract the protocol from the actual implementation and to infer the dongle's operations we ran a series of sessions on three different Nano dongles and one Nano S[4], compared the APDU command-response pairs, analysed the exchanged data and mapped it to the Bitcoin raw protocol (see Appendix A.1 for an example session). We concluded that during an active session four protocols may be executed:
(*a*) *Dongle Alive*: the initial communication when the dongle is plugged-in.
(*b*) *Setup*: wallet configuration and generation of the master keypair $\{sk_m, pk_m\}$.
(*c*) *Login*: user authentication to the dongle, and *vice versa*.
(*d*) *Payment*: processing of a payment transaction.
The *Dongle Alive* and *Login* protocols run once each time the dongle is connected to an active API. The *Payment* protocol repeats each time the user requests a payment. To proceed to a payment the user is not required to re-authenticate. The *Setup* protocol is executed once for initialising the wallet and each time an account restore is required; user authentication is its prerequisite. The dongle communicates with the API only when one of the four protocols are executed or when a firmware update is requested.

Commands Used During the Communication. Wallet communication consists of raw messages between the API and the dongle. To make the analysis readable we present the command-response messages in the form of $\mathtt{c}(p_1, p_2, \ldots, p_n) \rightarrow r_1, _2, \ldots, r_m$, which denotes that the API sends command `c` with parameters $p_1, p_2, \ldots, p_n, n > 0$ to the dongle; and the dongle responds with $r_1, r_2, \ldots, r_m, m \geq 0$. If $m = 0$ the dongle either replies with `OK` (success) or `error` (failure). Table 3 lists the communication primitives used to describe the protocols.

[3] Due to the sensitivity of the application we have not made our code publicly available. However, it can be made available to reviewers upon request.

[4] The protocol of Nano S is very similar to that of Nano, thus it was not necessary to test it in a different dongle.

command	meaning
`get_firmware_version()` $\rightarrow fV$	returns the dongle's firmware version fv
`get_wallet_public_key(`$bipDer_i$, $findex_i$, $lindex_i$) $\rightarrow pk_i$	given the number of bip derivations $bipDer_i$, the first index $findex_i$, the last index $lindex_i$, returns the public key pk_i
`get_device_attestation(`$blob$) $\rightarrow$ {Sig_{att},$attId$, $attDer$, $frwVer$, $modes$, $currentMode$}	returns the signature Sig_{att} of $blob$ which is the concatenated byte-string of firmware version $frmwVer$, with the private key sk_{att} the verification key parameters $attId, attDer$, the operation modes $modes$, the current mode $currentMode$ and $frmwVer$
`verify(`pin) $\rightarrow$ `OK`	sends the user's pin to the dongle; if correct, the dongle replies `OK`
`set_operation_mode(`$secFac$, $opMode$)$\rightarrow$ `OK`	sets the second factor authentication $secFac$ to true/false and the wallet operation mode $opMode$ to standard/relax/developer
`sign(`$bipDer_i$, $findex_i$, $lindex_i$, m) $\rightarrow$ `OK`	initialises the signature of the message m with the private key sk_i that corresponds to ($bipDer_i$, $findex_i$, $lindex_i$)
`sign(`pin) $\rightarrow Sig_m$	returns the signature Sig_m of message m with key the private key sk_i if the pin it provides is correct
`setup(`$pin, seed, genKey$) $\rightarrow$ `OK`	sets up a new user's pin, stores a new $seed$ and requests from the dongle to generate, $genKey$, a new $3DES_2$ key
`set_keyboard(`$chars$, $typeConf$) $\rightarrow$ `OK`	sets up the keymap characters $chars$ and the typing behaviour $typeConf$
`get_trusted_input(`X) $\rightarrow$ {Sig_t , oi, $amount_t$}	given X, where X is the raw structure (Table 2) for each previous output, returns the signature of each previous output Sig_t, the output index oi and $amount_t$ of the previous transaction t
`untrusted_hash_transaction_input_start(` Sig_t,oi, $amount$) $\rightarrow$ `OK`	streams the inputs, Sig_t, oi and $amount_t$ to the dongle using the raw structure (Table 2)
`untrusted_hash_transaction_input_finalize(` $addr_p$, $amount_p$, $fees_p$, $bipDer_c$, $findex_c$, $lindex_c$) $\rightarrow$ {c, $addr_p$, $amount_p$, $fees_p$, pk_c, $secFC$}	streams the outputs, payment address $addr_p$, payment amount $amount_p$, $fees_p$, and selects the key pk_c to which the change will be sent based according to its BIP32 parameters $bipDer_c$,$findex_c$,$lindex_c$. The command returns the change c, the change key pk_c, dongle's confirmation of add_p, $amount_p$, $fees_p$, and the characters of the address $secFC$ to be authenticated by the user
`untrusted_hash_sign (`$bipDer_i$, $findex_i$, $lindex_i$, $secFR$) $\rightarrow Sig_p$	returns the signature Sig_p of the transaction p with key sk_i given its BIP32 parameters $bipDer_i$, $findex_i$, $lindex_i$, iff $secFR$ is correct

Table 3: API Commands and their Meaning.

Keys that Appear During the Communication. We conclude that LEDGER wallets manage the following key types:
(*i*) $\{sk_{att}, pk_{att}\}$: predefined attestation keys, used for the dongle's firmware authentication and for setting up third-party hardware,
(*ii*) $\{sk_m, pk_m\}$: the master keypair from which all keys are derived,
(*iii*) $\{sk_i, pk_i\}$ pairs: transaction related keys, *i.e.*, keys$\{sk_r, pk_r\}$ for receiving funds and $\{sk_c, pk_c\}$ for transferring the change of a transaction. All keys, besides pk_r, are generated and stored dongle-side.
(*iv*) pk_{kp}: a symmetric key for the encryption/decryption of the wallet's key-pool. As most Bitcoin wallets do, LEDGER software maintains a key-pool of 100 randomly generated addresses: each time the wallet requires a new address it picks one from the key-pool which is then refilled. Based on the original Bitcoin client (*i.e.*, the Satoshi client) the key-pool gets encrypted (AES-256-CBC) with an entirely random master key [27]. This master key is encrypted with AES-256-CBC with another key derived from a SHA-512-hashed passphrase. In the original implementation, the user provides that passphrase when generating that key and each time he wishes to proceed to a trans-

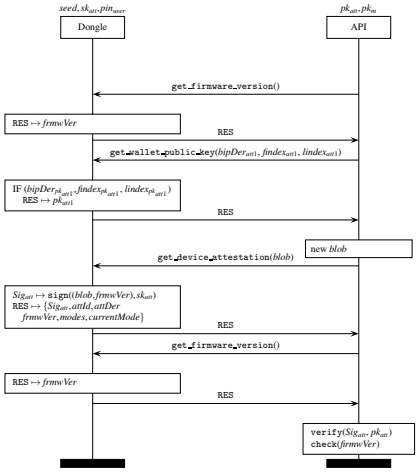

Fig. 1: The Nano Alive Protocol.

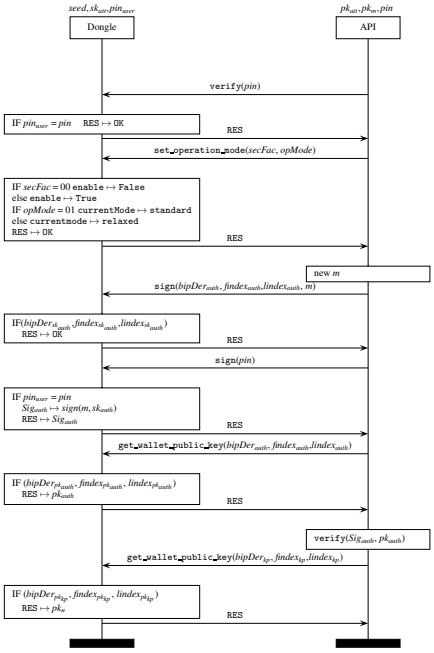

Fig. 2: The Nano Login Protocol.

action. LEDGER wallets use pk_{kp} as a passphrase to generate that encryption key.
(*v*) $\{sk_{auth}, pk_{auth}\}$: signature/verification keypair for the dongle-API authentication.

LEDGER dongles do not follow the common smart-card file structure: instead of supporting dedicated and elementary files, the keys are stored in a tree-like structure starting from the master key-pair and are referenced according to the corresponding BIP32 derivation parameters: (1) the number of derivations $bipDer$, (2) the first derivation index $findex$ and (3) the last derivation index, $lindex$.

Dongle Alive Protocol. *Ledger Nano:* The protocol consists of four message requests with which the API checks the integrity of the dongle's firmware through an attestation check: the API requests the dongle to sign a random $blob$ concatenated with the firmware version $frmwVer$ under a manufacturer key sk_{att}. The exact steps are: (*a*) The API retrieves the dongle's firmware version $frmwVer$. (*b*) The API retrieves pk_{att}. (*c*) The API sends $blob$ to the dongle and retrieves the signature Sig_{att} of the $blob$ concatenated to the firmware version $frmwVer$, the id of the attestation key $attId$, $frmwVer$ and the operation $modes$ and $currentMode$. (*d*) The API retrieves again $frmwVer$ and verifies Sig_{att}. The state transition diagram of the protocol can be found in Figure 1.

Ledger Nano S: The API retrieves pk_{att}, the dongle's firmware version $frmwVer$ in plaintext, sets the currency and retrieves the keys pk_{auth}, pk_{kp}. The Nano S protocol does not include the attestation authentication.

Login Protocol. *Ledger Nano:* The Login Protocol (Figure 2) establishes an authenticated session by which the user gains access to the dongle and, consequently, to the

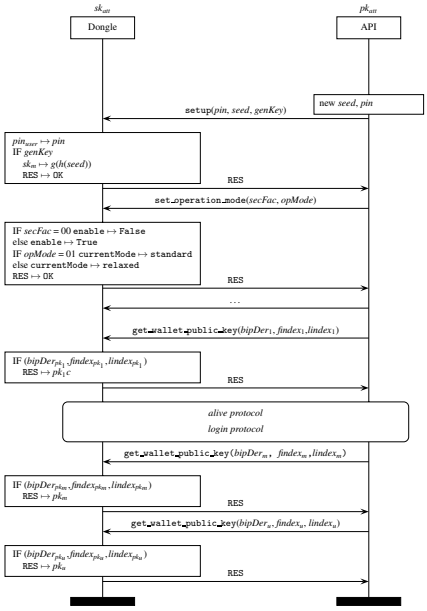

Fig. 3: The Nano Setup Protocol.

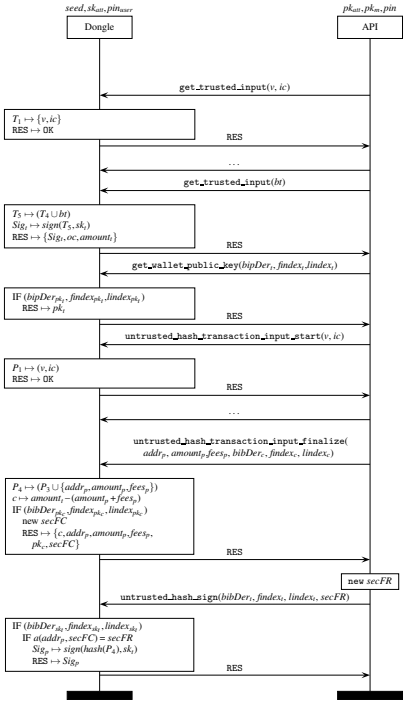

Fig. 4: The Nano Payment Protocol.

wallet. In contrary to Nano S in which no communication is involved (the user authenticates directly from the device's surface), the protocol consists of six messages, with the main operations being: (*a*) user pin verification, (*b*) dongle authenticity verification via a signature check, and (*c*) retrieval of wallet-related keys. The API also enables or disables second-factor authentication for payments and configures the wallet's operation modes. The supported modes are: (*i*) *standard*, the default, which allows standard Bitcoin scripts (addresses staring with 1) or P2PSH scripts (addresses staring with 3) and a single change address. At the beginning of the transaction the user is shown the amount to pay, the change, and any fees. (*ii*) *relaxed*, which allows arbitrary outputs to be authorised. At the beginning of a transaction the user is shown the amount to pay. (*iii*) *server*, allowing arbitrary outputs to be authorised but the transactions are controlled by a number of parameters.*e.g.*, maximum total of transactions. (*iv*) *developer*, allowing arbitrary data to be signed. The steps of the protocol are: (*a*) The API sends the user's pin to the dongle. (*b*) Upon pin verification the API sets the second factor authentication ($SecFac$) and wallet operation ($opMode$) modes. (*c*) The API requests the dongle to sign a random message m with key sk_{auth} and retrieves Sig_{auth} by sending pin. (*d*) The API retrieves pk_{auth} and verifies Sig_{auth}. (*e*) The API retrieves pk_{kp}.

Setup Protocol. *Ledger Nano:* The setup process begins API-side. After selecting a PIN, the user is given a 24-word passphrase which corresponds to the wallet's seed. After the user has confirmed the correct passphrase by providing the words that the API has requested, API-side initialisation is done. Then, the dongle-side setup begins.

The main operations of the *Setup* protocol (due to space limitation Figure 3 presents only the exchanged messages that are exploited by our attacks), are: user pin and seed initialisation, and the keyboard and operations mode setup. During initialisation, the API also retrieves the master public key pk_m, and the first derived public key pk_1. The message flow is the following: (*a*) The API sets up a new pin and $seed$ and requests the generation of $\{sk_m, pk_m\}$. (*b*) The API requests from the dongle to sign $frmwVer$ concatenated to a random $blob$ using the key sk_{att}. (*c*) The API verifies the pin. (*d*) The API retrieves pk_1. (*e*) The *Dongle Alive Protocol* takes place. (*f*) The *Login Protocol* takes place. (*g*) The API retrieves pk_m and some extra unidentified key k_u.

Ledger Nano S: Initialisation is performed dongle-side. The user is shown the 24-word mnemonic and the first time the dongle connects to the API, it sends pk_m.

Payment Protocol. Both Ledger Nano and Nano S use a second factor authentication mechanism to ensure that transactions are not tampered, with both implementations requiring the user's confirmation of the payment address. In Ledger Nano the second factor authentication is of the form of a challenge-response, based on a 58-character-pairs security card the user is provided with. Each time the dongle is requested to process a payment, it presents the user with a challenge $secFC$ consisting of four indexes of the payment address. The user responds to that challenge with the corresponding characters from the security card, $secFR$. Only if $secFR$ is correct, will the dongle continue processing the transaction. Nano S also requires user interaction to process a transaction: before signing the transaction it displays part of the payment address, the payment amount and the fees on its screen. Only if the user confirms the transaction data by pressing the OK button, the dongle will sign the signature.

LEDGER implements a proprietary *Segregated Witness* by enforcing the API to send a detailed description of the inputs before the payment processing: the API forms a pseudo transaction block which has only the inputs, and sends it to the dongle, through a set of `trusted_input` commands. The dongle parses the block (bytewise concatenation) and returns its signature Sig_i. When the API creates the actual transaction, it will use Sig_i to define the corresponding input.

Ledger Nano: The protocol, shown in Figure 4, is as follows: (*a*) The API sends to the dongle the available funds through sets of `get_trusted_input` commands. The inputs are sent in the form of pseudo transactions (following the specification in Table 2): one for each input. The number of `get_trusted_input` command sets is equal to the addresses ($t_i, i \geq 1$) with available funds. When the dongle has successfully received block t for a given input, it signs it and returns the signature Sig_t, the output index and the amount. (*b*) The API retrieves pk_t for input t. (*c*) The API creates the actual transaction block (Table 2), requested by the user, by sending the inputs Sig_t through sets of `untrusted_hash_transaction_input_start` commands, each set corresponding to a single input. Then, outputs, *i.e.*, the payment address $addr_p$, the payment amount $amount_p$, the fees $fees_p$ and the change key pk_c parameters ($bipDer_c$, $findex_c$, $lindex_c$), are sent via a `untrusted_hash_transaction_input_finalize` command. (*d*) The dongle calculates the remaining balance c, selects the authentication bytes $secFC$ sends back to the API a confirmation of the payment details, c, pk_c and $secFC$. (*e*) The API requests from the dongle to sign the transaction with sk_t by send-

ing the user's validation code, $secFR$. (*f*) The dongle checks $secFR$ against $secFC$ and $addr_p$ and, if it is correct, it computes and returns the transaction signature Sig_t.

Ledger Nano S: The *Payment* proceeds as presented in Figure 4 with a few differences: (*a*) The API starts the transaction by retrieving the balance address, pk_c, via a `get_wallet_public_key` command. (*b*) The API sends pk_c back to the dongle via the `untrusted_hash_transaction_input_finalize` command. (*c*) There is no second factor authentication asked by the dongle, or sent by the API.

4 Attacks

A Bitcoin wallet should provide high levels of security and privacy for the user, while also being easy to use. We therefore consider a wallet to be secure when it provides: (*a*) guarantees against tampering, (*b*) a secure environment for transaction processing, and (*c*) account privacy.

a. Direct wallet attacks	***b. Transaction attacks***
a.1 access to the master private key sk_m;	b.1 tamper the payment amount;
a.2 access to the key pool encryption key;	b.2 tamper the payment address;
a.3 unauthorised access to the wallet;	b.3 denial of service.
a.4 alter the wallet security properties.	***c. Account privacy attacks***
	c.1 account traceability.

Table 4: Attack categories

Our threat model assumes perfect cryptography and considers an adversary who has complete control over the communication layer: he can eavesdrop and manipulate the communication by deleting, inserting and altering the messages. We define the categories of possible threats to any Bitcoin wallets shown in Table 4.

4.1 Attacks in practice

We show how we were able to perform attacks from the APDU layer, by bypassing the restrictions of the API. Some attacks are passive, *i.e.*, they only require observing the communication channel; while others are active *i.e.*, involve relaying and altering the exchanged messages. Some example traces of the attacks can be found in Appendix.

a.1: Access to Master Private Key sk_m. Access to the wallet's seed s is synonymous to having access to sk_m. During the *Setup* protocol execution we were able to sniff s which was sent in plaintext from the API to the dongle. By using the BIP32 derivation function we regenerated sk_m and all children keys. The API having access to s and transmission of s in plaintext defeats the purpose of cold storage. The attacker may gain access to the *Setup* protocol, and consequently to s, by forcing the dongle's reinitialisation. *Mounting Attack a.1:* Given a valid pin p, a replay of the session {`verify`(p')$\rightarrow$ `error`, `verify`(p')$\rightarrow$ `error`, `verify`(p')$\rightarrow$ `error`} in which $p' \neq p$ results into the dongle entering a lock state and forcing re-initialisation. The attacker has now access to the *Setup* protocol and can either acquire s or inject his own seed s_a.

a.2: Access to Key-Pool Encryption Key. Unauthorized access to key-pool implies loss of privacy and account treacability as the adversary gains insight on the addresses that the account uses/has used. During a *Login* session the passphrase that is used to

create the key-pool key, which is the key pk_{kp}, is transmitted in plaintext after a `get_wallet_public_key` command.

a.3: Unauthorised Access to the Wallet. A general requirement in Bitcoin wallets is to be used only by users that have the credentials, *e.g.*, the pin. Our analysis showed that at each *Login* protocol execution the pin is sent in plaintext, (though only in the LEDGER Nano case), via a `verify` command, making the pin vulnerable to eavesdropping.

a.4: Alter the Wallet Security Properties. A second factor authentication mechanism secures each transaction: the user has to verify random characters of the payment address. The following attack changes the security parameters of the dongle and disables that mechanism. *Mounting Attack a.4:* Perform the following steps: (a) Replay a legit *Setup* session, {`setup(`p`,`s`)`$\rightarrow$`OK`, ..., `set_operation_mode(`$enable$`,` $standard$`)`$\rightarrow$`OK`, ... } (Figure 3), and apply the substitutions ($\mapsto$): `set_operation_mode(enable` $\mapsto$ `disable, standard` $\mapsto$ `relaxed)`. (b) In each *Login* session, (Figure 2), replay the communication by applying the substitutions ($\mapsto$): `set_operation(enable` $\mapsto$ `disable, standard` $\mapsto$ `relaxed)`. (c) In each *Payment* session, (Figure 4), replay the communication and apply the substitutions ($\mapsto$): i) in `untrusted_transaction_input_hash_finalize`: `response(`c,$addr_p$, $amount_p$, pk_c, `no` $\mapsto$ $secFC$) where `no` is the card's response that no second authentication is required, and $secFC$ are four random characters of the payment address $addr_p$. ii) `untrusted_has_sign(`sk_params`,` $secFR \mapsto$ `no)` where $secFR$ is the user's input to $secFC$ and `no` declares that no secondary authentication took place.

a.4: Learning the Security Card. If the second factor authentication mechanism is enabled, each transaction requires the user's input according to a security card. The dongle requests four characters of the payment address to be verified by providing their mappings of the security card (58 hexadecimal characters that encode the letters A-W, a-w and the numbers 0-9). Each *Payment* session can reveal four new mappings. For this, 1. the adversary alters $secFC \mapsto secFC'$ in favour of the character mappings he does not yet know but that will allow him to correctly compute the response to the challenge, 2. the adversary returns to the dongle the correct $secFR$ according to the original challenge $secFC$. In this way the adversary will learn four new characters in each *Payment* protocol execution. And so, after 15 legitimate user initiated payments, the attacker will have learned all characters of the security card.

b.1-b.2: Transaction Attacks. Given a *Payment* session an adversary can (a) redirect the payment destination: $addr_p$ and (b) tamper the payment amount: $amount_p$ by altering the exchanged messages. *Mounting attack b.1-b.2:* (a) to redirect the payment destination apply the following substitutions ($\mapsto$): `untrusted_transaction_input_hash_finalize(` $addr_p \mapsto addr'_p$, $amount_p$,$fees_p$, pk_c $parameters$) $\rightarrow$ `response(`c, $addr_a \mapsto addr_p$, $amount_p$, pk_c, $secFC$). In the command data the original payment address $addr_p$ is substituted by the attacker's add'_p. The response is also relayed so that it contains the original address.
(b) to tamper the payment amount one should apply the following substitutions ($\mapsto$): `untrusted_transaction_input_hash_finalize(` $addr_p$, $amount_p \mapsto amount'_p$, $fees_p$, pk_c $parameters$) $\rightarrow$ `response(`$c' \mapsto c$, $addr_p$, $amount'_p \mapsto amount_p$, pk_c, $secFC$). In the command data the the original payment amount $amount_p$ is substituted by the attacker's $amount'_p$ whereas in the response data $amount'_p$ is changed back to

$amount_p$ and the remaining funds c' is changed to the amount that would result after the original payment amount. This attack combined with either of the two a.4-type attacks allows an attacker to have any transaction signed by the dongle, even without knowledge of the PIN or the master secret key.

b.3: Denial of Service. DoS attacks that target specific Bitcoin Wallet users have become viral, *e.g.*, the DoS attacks on the BitGo wallets that left many users unable to use their funds. Such attacks target the wallet's server and consist of sending a huge amount of requests. Though out of our scope, in the LEDGER wallet side of things, DDoS attacks could also be mounted from the APDU layer by tampering the transaction data in a way that either the dongle cannot interpret, or that the transaction cannot be verified.

c.1: Account Traceability. Bitcoin is associated with anonymity and is often used by users who want their actions to be unlinkable. Each transaction results to the generation of a new key to avoid reusability of old addresses. In HD wallets, like LEDGER, all public keys are derived from the master public key pk_m with the formula $pk_i = f(pk_m + hash(i, pk_m))$ where i is the child key index and f the generator function. As such, access to pk_m equals to access to all pk_i keys and thereby the account becomes traceable. pk_m can be obtained by eavesdropping a *Setup* session (which can be enforced with Attack 4.1) as in both Nano and Nano S is transmitted in plaintext. However, the account's activities are also traceable by just eavesdropping the *Payment* sessions: at least one pk_r (an address with available funds) and probably one change address pk_c (if their available change) are revealed.

Generality of the Attacks. The purpose of our work is to show that it is possible to attack Bitcoin hardware wallets via the low-level communication. The threat model we present is hardware/software independent and applicable to all available Bitcoin wallets. The attacks on the LEDGER wallets aim to prove that Bitcoin transactions are vulnerable, even if tamper-resistant hardware such as smart-cards are incorporated. Our work showcases how the API restrictions can be bypassed by relaying the hardware communication. The same attacks, adapted to meet the criteria of each hardware, can be applied to every wallet that does not use a secure communication channel *i.e.*, Trezor and Keepkey. All hardware wallets follow the same abstraction of the *Payment* protocol; any plaintext communication is prone to attacks b.1-b.2. Althought they incorporate a second factor authentication mechanism by enforcing the user's verification of the payment data, previous studies have shown that a significant average of 15% of such verification is usually erroneous.

The privacy issues we address for the LEDGER wallets is an aspect that reflects to all BIP32 wallets, especially to those that that do not communicate in a secure way. Currently, all hardware wallets[5] transmit the public keys (including the master public key) in the clear: eavesdropping a single session reveals at least two public keys: the address with available funds and the address that the remaining balance will be sent to. Also, whenever the hardware connects to a fresh API, the master public key pk_m is sent in the clear. Access to that key implies access to all children public keys, which makes eavesdropping that single session sufficient to track the account's transactions. In any case, whether the adversary has access to pk_m or to its children pk_i the the flow of the funds of the given account is linkable.

[5] Apart from Digital BitBox whose specifications are not available publicly.

5 A Lightweight Fix of the Protocols

The LEDGER wallets, as all other hardware wallets not using a secure communication channel, fail to prevent MitM attacks. All transaction data is sent in the clear, making the wallet vulnerable to attacks and account linkability. Encrypting the entire communication would be an obvious solution to that. However, such strategy requires computational power, and possible changes to the security architecture of the current wallets. Additional delays to the transaction processing would be another trade-off. Instead we propose the symmetric encryption of specific communication parts: those that are prone to attacks with respect to our threat model. Table 5 summarises what LEDGER wallet data need to be protected to defend against which attacks. Our fix consists of three components: 1. the secure pre-setup phase, 2. the authentication and session key establishment protocol, 3. encryption of sensitive parts.

data	a.1	a.2	a.3	a.4	b.1	b.2	b.3	c.1
s	✓	✓	✓	✓	✓	✓	✓	✓
pin	×	×	✓	×	×	×	×	×
$secFC, secFR$	×	×	×	✓	×	✓	✓	×
$opMode$	×	×	×	✓	×	✓	✓	×
$addr_p$, $amount_p$,$fees_p$,c,pk_c	×	×	×	×	✓	✓	✓	✓
$pk\_\{m, m+1.., m+n\}$	×	✓	×	×	×	×	×	✓

Table 5: LEDGER Data and the Corresponding Attacks.

Secure Environment for the PIN Exchange. The PIN needs to be entered in the hardware before the initialisation of the wallet as the PIN is then used to derive the cryptographic keys to protect the interactions between the dongle and the API. This process must proceed in a secure offline environment. This can be achieved either by entering the PIN directly on the trusted user interface of the device (if it is an HID wallet); or by setting up the PIN on an air-gapped machine, *e.g.* using a live OS on a USB stick which will ensure that the OS has and will never be connected to the Internet.

Authentication and session key establishment. This protocol gets executed every time the API establishes a new session with the dongle. It is responsible for the API/hardware authentication and the establishment of a fresh session key. A new session is established whenever the hardware connects to an active API. For the key establishment we propose Password Authenticated Key Exchange by Juggling protocol (j-PAKE) [14] which allows bootstrapping high entropy keys from the low-entropy user's PIN. In that way, we avoid storing secret data API side, ensure that fresh keys are used in each session and guarantee the user's presence at that session. In addition, the j-PAKE protocol allows zero knowledge proof of the PIN which satisfies the authentication prerequisities of the session. Finally j-PAKE provides guarantees against off-line and on-line dictionary attacks and it satisfies the forward secrecy and known-key security requirements. J-PAKE, like the Diffie-Hellman key exchange, uses ephemeral values but proceeds in an additional round in which combines them with the user's PIN and makes certain randomisation vectors vanish.

Encryption of sensitive data. Once the session key is established slightly modified versions of the four LEDGER protocols (*Alive*, *Login*, *Setup*, and *Payment*) can be executed. The four new protocols are derived from the original Ledger protocols as follows. First a session identifier is established for each execution of each of these protocols.

This will be generated dongle side, and transmitted to the API in plaintext. The session identifier does not need to be confidential, but will need to be fresh and generated by the dongle to avoid replay attacks. Then dongle and API execute the original protocol but encrypting under the current session key the sensitive data identified previously (Table 5). The computed ciphertexts will all include the established session identifier. A Message Authentication Code (MAC) is further computed and concatenated to the chiphertext. The other party will then be able to decrypt and verify the encrypted parts.

6 Discussion

Although the security of financial related hardware in other areas has always attracted a lot of attention, eg., the Chip and PIN systems [23], Bitcoin-related hardware has not been extensively studied before (apart from [9]). Relying on the high levels of security that the Bitcoin protocol offers is not enough to guarantee safe transactions. Lack of a standard that defines the properties of the Bitcoin wallets leads to security misconceptions and ad-hoc implementations that hide vulnerabilities. Our work, to the best of our knowledge, is the first effort to address security aspects of Bitcoin wallets and stress the importance of securing the implementations of low-level communications. We chose to analyse smart-card based wallets as they are perceived to be the most secure and tamper resilient means for key management. However, the core idea of the attacks is general and applies to other hardware wallets of different technology.

In this paper we extract and analyse the protocols that are hidden behind the LEDGER wallets, the only available smart-card based solutions. Our work includes the analysis of both standard and HID dongles. We identify and categorise all possible vulnerabilities for Bitcoin wallets and we introduce a general threat model. We then use that model to analyse the LEDGER protocols. Our work concluded that the LEDGER implementations are vulnerable to a set of attacks that target the wallet itself as well as the Bitcoin transactions. Finally, we propose a lightweight fix, based on the j-PAKE protocol, which can easily be adapted by any wallet and efficiently prevents any active or passive attack. Attacking the LEDGER wallets is just an example, whereas the same methodology can be easily adopted in other technologies. Our work does not aim at proving the specific wallets insecure, but rather to showcase the importance of ensuring a secure low-level implementation even if the higher levels provide guarantees.

References

1. E. Androulaki, G. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in bitcoin. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 34–51, 2013.
2. T. Bamert, C. Decker, R. Wattenhofer, and S. Welten. *BlueWallet: The Secure Bitcoin Wallet*, pages 65–80. 2014.
3. S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better - how to make bitcoin a better currency. In *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, pages 399–414, 2012.

4. E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*, pages 513–525, 1997.
5. Bitcoin ewallet vanishes from internet. `http://www.tribbleagency.com/?p=8133`.
6. Bitcoin Protocol Documentation. `https://en.bitcoin.it/wiki/Protocol%5Fdocumentation`.
7. Bitcoinmagazine. `https://bitcoinmagazine.com/articles/ozcoin-hacked-stolen-funds-seized-and-returned-by-strongcoin-1366822516`, 2013.
8. C. Bozzato, R. Focardi, F. Palmarini, and G. Steel. Apdu-level attacks in pkcs# 11 devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 97–117, 2016.
9. J. Datko, C. Quartier, and K. Belyayev. Breaking bitcoin hardware wallets. DEFCON 2017.
10. G. De Koning Gans and J. De Ruiter. The smartlogic tool: Analysing and testing smart card protocols. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 864–871, 2012.
11. C. Decker and R. Wattenhofer. Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.
12. D. Genkin, A. Shamir, and E. Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*, pages 444–461, 2014.
13. A. Gkaniatsou, F. McNeill, A. Bundy, G. Steel, R. Focardi, and C. Bozzato. Getting to know your card: reverse-engineering the smart-card application protocol data unit. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 441–450, 2015.
14. F. Hao and P. Ryan. *J-PAKE: Authenticated Key Exchange without PKI*, pages 192–206. 2010.
15. J. Herrera-Joancomartí. Research and challenges on bitcoin anonymity. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance - 9th International Workshop, DPM 2014, 7th International Workshop, SETOP 2014, and 3rd International Workshop, QASA 2014, Wroclaw, Poland, September 10-11, 2014. Revised Selected Papers*, pages 3–16, 2014.
16. S. Higgins. `http://www.coindesk.com/unconfirmed-report-5-million-bitstamp-bitcoin-exchange`, 2015.
17. H.-C. Hsiao, Y.-H. Lin, A. Studer, C. Studer, K.-H. Wang, H. Kikuchi, A. Perrig, H.-M. Sun, and B.-Y. Yang. A study of user-friendly hash comparison schemes. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 105–114. IEEE, 2009.
18. D. Y. Huang, H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. McCoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko. Botcoin: Monetizing stolen cycles. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.
19. G. O. Karame, E. Androulaki, and S. Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 906–917, 2012.
20. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397, 1999.
21. P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113, 1996.
22. I.-K. Lim, Y.-H. Kim, J.-G. Lee, J.-P. Lee, H. Nam-Gung, and J.-K. Lee. *The Analysis and Countermeasures on Security Breach of Bitcoin*, pages 720–732. 2014.
23. S. J. Murdoch, S. Drimer, R. J. Anderson, and M. Bond. Chip and PIN is broken. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 433–446, 2010.
24. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.

25. K. Poulsen. New malware steals your bitcoin. https://www.wired.com/2011/06/bitcoin-malware, 2011.
26. M. Rosenfeld. Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009, 2014.
27. The Bitcoin Wiki. https://en.bitcoin.it/wiki/Wallet%5Fencryption.
28. The Bitcoin Wiki. https://en.bitcoin.it/wiki, 2014.
29. M. Turuani, T. Voegtlin, and M. Rusinowitch. Automated verification of electrum wallet. In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC*, pages 27–42, 2016.
30. E. Uzun, K. Karvonen, and N. Asokan. Usability analysis of secure pairing methods. In *International Conference on Financial Cryptography and Data Security*, pages 307–324. Springer, 2007.
31. P. Wuille. Dealing with maellability. *Online specification for BIP62*, 2014.
32. P. Wuille. Hierarchical deterministic wallets. *Online specification for BIP32*, 2017.

A Appendix

A.1 Example Communication Trace

As an example, we provide the trace that was generated for the following transaction:

Transaction Id	92d30a91b45d6ab528af12f3a9c0701e01f67348a257ed50362439a2ee8274e7
Input addresses	**1** 113biVTVQk73Eem1UYYn9YcrPVrxp6xeVc **2** 15DpocdQpwXeUp9Ccf2Nz9AQ9jKp9U5VdZ
Payment address	1GocNQ4Q8BtzacpHQiGLWk9vNppoq6Lh8W
Payment amount	0.00813844
Change address	1PmXm9UcAgDBp5i3SvqD3SfdKChfWthH4W

We only provide the traces of the commands of Figure 4 so as not to overwhelm.

1. get_trusted_input: e042000009000000010100000001
2. response: 9000
3. get_trusted_input: e04280000400000000
4. response: 32008ed5f038879105a5778cdacee02ca43f21bcbbd66cd647add3db69dd3222b9c3968d000
   0000078710d00000000009132801b579e659b
5. get_wallet_public_key: e040000015058000002c800000008000000000000000000000000c
6. response: 410441ec4b255d40010284f117d8105456a268cd9536ca5ca3d3016bf6d21902e5dc4bf9b22
   4b5cb2379b5c2b4a47044862d42c6e5b14daf22939fec8023c83ac519223131336269565456516b37334
   5656d315559596e3959637250567278703678655663da55cec9398694400832d6af2426c057addc73438
   efa016f6f9232735ee6b1a8
7. get_wallet_public_key: e040000015058000002c800000008000000000000000100000012
8. response: 41043f07a649a72651f10d5728b7f848ee879fb3b263ddd653b51b563a051f138fa3e35f5f6
   d794a2621fbf0493d6af5c2b300734086fa0ebbe411f11017b1989bdd22313544706f636451707758655
   57039436366324e7a394151396a4b7039553556645a9bf32153ef7f646d1d1991382932bc915d671ddc3
   640ef8da3eb54877191e559
9. untrusted_hash_transaction_input_start: e0440000050100000002
10. response: 9000
11. untrusted_hash_transaction_input_finalize: e04602004822314 76f634e5134513842747a616370
    485169474c576b39764e70706f71364c6838570000000000000d6d8000000000000045080580000002c8000
    0000800000000000000100000013
12. response: 4502d8d60000000000001976a914ad5a8ba5325b4b836c49b09797cbb83744a7a2f588ac146
    b0c00000000001976a914f9bebf6735e688877e409cd494ad820b344dd76e88ac03040405121e47646f8
    13e5dfd4fbc72e6698cc40a67a980bccbe7881c2e40ac6fec4fbcda20d980ec3a67445e48dad870ee58d
    006745fdf953138be5fb0570e679f512c36ed
13. untrusted_hash_sign: e04800001f058000002c800000008000000000000000000000c040406060200
    00000001
14. response: 3044022033128d0d576487e2e0c5892c0915564a6a5f119e698c033262d660527943a16d022
    009caa037703d9a3dbf7eec4cecca08bf33b3b9a18ef929a810f8faf6ab0f1c7a01

Steps	APDU traces
Block the dongle	• verify(p'): 02200000433333333 • verify(p'): 02200000433333333 • verify(p'): 02200000433333333
Replay a Setup Session	• setup(p, s): e02000004c020a00050431343234 00408c3937fafb22e5f4979e90afe0b912cc05d92b9910c622887f61b30d9814f714df2dd5ada8cc5cd663e998dec1cc55915377352cf6949a20ba444039219efd6900 • set_keyboard: e028000077000000000000000000000000760f00d4ffffffc7000000782c1e3420212224342627252e362d3738271e1f202122232425263333362e37381f0405060708090a0b0c0d0e0f101112131415161718191a1b1c1d2f3130232d350405060708090a0b0c0d0e0f1011121314151617181191a1b1c1d2f313035 • get_device_attestation: e0c200000861255ccee7f8c72d • set_operation: e02600000102 ...

Table 6: Attack a.4: Trace of disabling the second factor authentication during *Setup*

A.2 Active Attacks

a.4: Alter the Wallet Security Properties. The attack requires sending the wrong pin p' three consecutive times and then tampering the `set_operation` command. A sample trace with the breakdown of the steps and their corresponding commands is given in Table 6; we underline the important pieces of the exchahge.

b.1-.b2: Transaction Attacks. The structure of `untrusted_hash_transaction_input_finalize` is:

command	e046020048
length of payment address	22
payment address $addr_p$	314e3371757233596565334b664e74436a4677756e346f366f4c324478686747796f
payment amount $amount_p$	0000000000005305
fees $fees_p$	0000000000001d60
change address BIP32 parameters	058000002c800000008000000000000001000000
second authentication status (true/false)	02

and the structure of the response data that we are interested in is:

payment amount $amount_p$	03b1000000000000
hash160 of $addr_p$	f1253f0463e5877c5e8bb3f34e7abfb335023ee1
change c	0553000000000000
hash160 change address $addr_c$	e6e44d66125327341d6abb71e0702a4ea0537437

Depending on the attack we want to perform the corresponding data part needs to be altered. For example, to change the payment address from `163WPEeTHjvFsUfx1UbDPXK92eRmqXQrGA` to `113biVTVQk73Eem1UYYn9YcrPVrxp6xeVc`, we tamper the original command:

e046020048223136335750456554486a7646735566783155624450584b393265526d7158517247410000000000027100000000000001a9a058000002c800000080000000000000100000000

to the command:

e046020048223131336269565456516b373345656d315559596e3959637250567278703678655663000000000002710000000000001a9a058000002c800000080000000000000100000000

where we underline the relevant parts; similarly for the response.

Learning the Security Card. The attacker gains access to the keycard mappings, $secFR$, via the `untrusted_hash_sign` command, *e.g.*, e04800001f058000002c8000000080000000000000000001040f090a02 0000000001.